**REST API**

**Topic:** Online shop.

**Author:** Letunovsky Igor

**Requirements.**

**Functional.**

The API supports the creation of new users, logging, editing user information.

Users are divided by their roles. For simplicity, there are two roles: admins and users.

Admins can create, update and delete shop items and users.

Users can view items as well as use pagination, filtering and sorting.

Users can add and delete items to their cart.

Users can proceed with payment with all items from their cart, initiating an order.

Users can set rating and leave comments for every product.

**Non-functional.**

Authorization via JWT tokens.

Applicable method should be paginated.

Caching is used.

Richardson Maturity Model is applied at level 3 along with HATEAOS.

**Entity models**

 **User**

* user\_id (int)
* username (String)
* password (String)
* email (String)
* role (Enum)
* created\_at (Timestamp)
* updated\_at (Timestamp)

 **Item**

* item\_id (int)
* name (String)
* description (Text)
* price (Decimal)
* stock (int)
* created\_at (Timestamp)
* updated\_at (Timestamp)

 **Cart**

* cart\_id (int)
* user\_id (int)
* created\_at (Timestamp)
* updated\_at (Timestamp)
* total\_price (Decimal)

 **CartItem**

* cart\_item\_id (int)
* cart\_id (int)
* item\_id (int)
* quantity (int)
* price (Decimal)

 **Rating**

* rating\_id (int)
* user\_id (int)
* item\_id (int)
* rating (int)
* comment (Text)
* created\_at (Timestamp)

**Relationships:  
 One-to-many:**

* User has many ratings.
* Item has many ratings.

**One-to-one:**

* User has one cart.

**Many-to-many:**

* Cart has many cart items.

**Operations:** All entities must support CRUD operations, as stated in the functional requirements section.

**REST API Description**

**Collections:**

* **/users** – managing users
* **/items** – managing items
* **/carts –** managing carts

**Operations:**

**/items**

1. Create new item in the shop

*Method:* POST /items  
 *Request:* all item attributes.  
 *Response:* 201 Created.  
 *Roles:* admins.  
 *Possible errors:* 400 Bad Request (invalid params), 401 Unauthorized (not logged in), 403 Forbidden (low role permissions), 409 Conflict (if item id is reinserted), 422 Unprocessable Entity (wrong types etc.), 500 Internal Server Error, 503 Service Unavailable, 429 Too Many Requests.

*Example:*

POST /items  
Authorization: Bearer \*token\*  
Content-type: application/json  
{  
\*\* here goes a json object with all filled field of item model \*\*  
}

And then, the 201 Created response with some json message.

2. Get all items – tied to shop catalogue page.

*Method:* GET /items  
 *Params:* page, limit (numeric)  
 *Response:* 200 OK, 304 Not Modified (cached)  
 *Roles:* any. *Possible errors:* 400 Bad Request, 500 Internal Server Error, 503 Service Unavailable, 429 Too Many Requests.

*Example:*

*Request:*

GET /items?page=1&limit=20  
Accept: application/json  
- or  
If-None-Match: “main-page-1”

*Response:*

200 OK  
Content-Type: application/json  
Cache-Control: max-age=666, public  
ETag: “main-page-1”  
{  
“page”:1,  
“limit”:20,  
“total”:328,  
“data”: [  
 {  
 \*\* item json \*\*  
 },  
 {  
 . . .  
 },  
 …,  
 …,  
 {  
 …  
 }  
],  
“\_links”:{  
 “self”:”/items&page=1&limit=20”,  
 “next”:”/items&page=2&limit=20”,  
 “prev”:null  
 }  
}

Caching is implemented using “Cache-Conrol”, “ETag” and “If-None-Match”. ETag can be considered as the id of the cached data. If-None-Match finds if there cached data with this ETag (this determines if 200 OK or 304 Not Modified is returned). Cache-Control enables and sets lifetime for cached data. Links field is responsible for meeting HATEAOS.

3. Get item by id – tied to viewing page of specific item

*Method:* GET /items/{id}  
 *Response:* 200 OK  
 *Roles:* any.  
 *Possible errors:* 400 Bad Request, 404 Not Found, 500 Internal Server Error, 503 Service Unavailable, 429 Too Many Requests.

*Example:*

*Request:*

GET /items/2  
Accept: application/json

*Response:*

200 OK  
Content-type: application/json  
Cache-Control: max-age=666, public  
ETag: “item-page-2”  
{  
\*\* item json object \*\*  
“\_links”: {  
 “self”: “/items/2”,  
 “related”:”/items?category=Food”  
 }  
}

4. Filtering

*Method:* GET /items  
 *Params:* page, limit, (numeric) category, seller, region (str)  
 *Response:* 200 OK, 304 Not Modified.  
 *Roles:* any.  
 *Possible errors:* 400 Bad Request, 500 Internal Server Error, 503 Service Unavailable, 429 Too Many Requests.

*Example:*

*Request:*

GET /items?category=Food&seller=Walmart&region=NA&page=1&limit=20  
Accept: application/json  
- or  
If-None-Match: “main-page-1-food-walmart-na”

*Response:*

200 OK  
Content-type: application/json  
Cache-Control: max-age=666, public  
ETag: “main-page-1-food-walmart-na”  
{  
“page”:1,  
“limit”:20,  
“total”:444,  
“data”:[  
 {  
 \*item object\*  
 },  
 …  
 {  
 \*item object\*  
 }  
],  
“\_links”:{  
 “self”:” /items?category=Food&seller=Walmart&region=NA&page=1&limit=20”,  
 “next”:null,  
 “prev”:null  
 }  
}

5. Update an item

*Method:* PUT /items/{id}  
 *Request:* full item object.  
 *Response:* 200 OK  
 *Roles:* admin  
 *Possible Errors:* 400 Bad Request, 401 Unauthorized, 403 Forbidden, 404 Not Found, 422 Unprocessable Entity (wrong types etc.), 500 Internal Server Error, 503 Service Unavailable, 429 Too Many Requests.

*Example:*

*Request:*

PUT /items/2  
Authorization: Bearer \*token\*  
Content-type: application/json  
{  
\*\* full items json object \*\*  
}

*Response:*

200 OK  
Content-type: application/json  
{  
\*\* full updated items object \*\*,  
“\_links”:{  
 “self”:”/items/2”,  
 “related”:”/items?shop=Walmart”  
 }  
}

6. Delete an item

*Method:* DELETE /items/{id}  
 *Response:* 204 No Content or 200 OK  
 *Roles:* admin  
 *Possible errors:* 400 Bad Request, 401 Unauthorized, 403 Forbidden, 404 Not Found, 500 Internal Server Error, 503 Service Unavailable, 429 Too Many Requests.

*Example:*

*Request:*

DELETE /items/2  
Authorization: Bearer \*token\*  
Accept: application/json

*Response:*

May be an object of deleted item, may be only 204 or 200 code.

**/users**

1. Create new user – tied to register

*Method:* POST /users *Request:* \*\* full users json object \*\* *Response:* 201 Created. *Roles:* any. *Possible errors:* 400 Bad Request, 403 Forbidden (client did not meet requirements – IP, etc.), 409 Conflict, 422 Unprocessable Entity (wrong types etc.), 500 Internal Server Error, 503 Service Unavailable, 429 Too Many Requests.

*Example:* Holds the same CRUD logic as in **/users** section. Links to self are provided (profile).

2. Get the list of users

*Method:* GET /users *Params:* page, limit (numeric), role (str) *Response:* 200 OK, 304 Not Modified *Roles:* admin (because in online shop there is no access to other users). *Possible errors:* 400 Bad Request, 401 Unauthorized, 403 Forbidden, 500 Internal Server Error, 503 Service Unavailable, 429 Too Many Requests.

*Example:* Holds the same CRUD logic as in **/users** section. Links to self are provided, along with next and prev if possible. Sensitive data is not sent with the response. (*/users?role=admin&page=1&limit=20*).

3. Get a user info – tied to user profiles

*Method:* GET /users/{id} *Response:* 200 OK, 304 Not Modified *Roles:* admin and the owner if the id. *Possible errors:* 400 Bad Request, 401 Unauthorized, 403 Forbidden, 404 Not Found, 500 Internal Server Error, 503 Service Unavailable, 429 Too Many Requests.

*Example:* Holds the same CRUD logic as in **/users** section. Links to self are provided. Sensitive data is not sent with the response.

4. Update user – tied to edit user profiles

*Method:* PUT /users/{id} *Request:* \*\* full users entity \*\* *Response:* 200 OK *Roles:* admin and the owner of the id. *Possible errors:* 400 Bad Request, 401 Unauthorized, 403 Forbidden, 404 Not Found, 422 Unprocessable Entity (wrong types etc.), 500 Internal Server Error, 503 Service Unavailable, 429 Too Many Requests.

*Example:* Holds the same CRUD logic as in **/users** section. Links to self are provided. Sensitive data is not sent with the response.

5. Delete a user

*Method:* DELETE /users/{id} *Response:* 204 No Content or 200 OK *Roles:* admin only. *Possible errors:* 400 Bad Request, 401 Unauthorized, 403 Forbidden, 404 Not Found, 500 Internal Server Error, 503 Service Unavailable, 429 Too Many Requests.

*Example:* Holds the same CRUD logic as in **/users** section.

**/carts**

1. Create a new cart

*Method:* POST /carts *Request:* user\_id only. *Response:* 201 Created *Roles:* authorized. *Possible errors:* 400 Bad Request, 401 Unauthorized, 403 Forbidden, 409 Conflict, 422 Unprocessable Entity (wrong types etc.), 500 Internal Server Error, 503 Service Unavailable, 429 Too Many Requests.

*Example:*

*Request:*

POST /carts  
Authorization: Bearer \*token\*  
Content-type: application/json  
{  
“user\_id”:38  
}

*Response:*

201 Created  
Content-type: application/json  
{  
“cart\_id”:1,  
“user\_id”:123,  
“created\_date”: date,  
“updated\_date”: same as created,  
“total\_price”: 0  
}

2. Get cart, along with all items tied to it though cartitems table

The same action is available for all carts for admins. This get request by id is more common.

*Method:* GET /carts/{user\_id} *Response:* 200 OK *Roles:* admin and the owner if the id. *Possible errors:* 400 Bad Request, 401 Unauthorized, 403 Forbidden, 404 Not Found, 500 Internal Server Error, 503 Service Unavailable, 429 Too Many Requests.

*Example:*

*Request:*

GET /carts/38  
Authorization: Bearer \*token\*  
Accept: application/json  
- or  
If-None-Match: “cart38”

*Response:*

200 OK  
Content-type: application/json  
Cache-Control: max-age=666, public  
ETag: “cart38”  
{  
“cart\_id”:354,  
“user\_id”:38,  
“created\_date”:date,  
“updated\_date”:date,  
“total\_price”:23132,  
“cart\_items”:[  
 {  
 \*item object got from the bridge cart item table\*  
 },  
 …  
 {  
 \*item object got from the bridge cart item table \*  
 }  
],  
“\_links”:{  
 “self”:”/carts/38”  
 }  
}

3. Place item into the cart

*Method:* POST /carts/{user\_id} *Request:* item\_id *Response:* 201 Created *Roles:* admin and the owner if the id. *Possible errors:* 400 Bad Request, 401 Unauthorized, 403 Forbidden, 409 Conflict, 422 Unprocessable Entity (wrong types etc.), 500 Internal Server Error, 503 Service Unavailable, 429 Too Many Requests.

*Example:*

*Request:*

POST /carts/44  
Authorization: Bearer \*token\*  
Content-type: application/json  
{  
“item\_id”:38  
}

*Response:*

201 Created  
Content-type: application/json  
{  
“cart\_item\_id”:21,  
“cart\_id”:321,  
“item\_id”:38,  
“quantity”:3,  
“price”: 777  
}

4. Delete item from the cart

*Method:* DELETE /carts/{uder\_id} *Request:* cart\_item\_id *Response:* 204 No Content or 200 OK *Roles:* admin or the owner of the id. *Possible errors:* 400 Bad Request, 401 Unauthorized, 403 Forbidden, 404 Not Found, 500 Internal Server Error, 503 Service Unavailable, 429 Too Many Requests.

*Example:*   
Cart is found by user id, then the item from cart idem bridge table is deletes, as the id is passed as request.

5. Edit item in the cart (quantity)

*Method:* PUT /carts/38  
 *Request:* cart\_item\_id and quantity *Response:* 200 OK *Roles:* admin or the owner of the id. *Possible errors:* 400 Bad Request, 401 Unauthorized, 403 Forbidden, 404 Not Found, 422 Unprocessable Entity (wrong types etc.), 500 Internal Server Error, 503 Service Unavailable, 429 Too Many Requests.

*Example:*

Editing follows the same CRUD logic as in previous examples, but take only cart\_item\_id and quantity to be updated. Link attributes to self are provided in the response.

**~~/rate~~**

Rating is no separate endpoint, but a subroute for either user of item routes. It may support the same CRUD actions along with pagination, sorting and filtering, like in the **/users** section.

**Conclusion**  
Online shop take a lot of resources to develop, and of course this is not all, many details are omitted and overall logic is oversimplified. However, this maps the basic feeling of the REST API and how it should work together.  
All requirements are considered. Pagination, filtering and authorization are applied where necessary. Links attribute to self and other point are provided to meet HATEAOS. Caching is also implemented in several examples where large GET requests are processed.